1. 在ShuffleMapTask的时候最终会调用ShuffleRDD.compute方法计算，源码如下：
2. 最终会创建BlockStoreShuffleReader

**override def** compute(split: Partition, context: TaskContext): Iterator[(K, C)] = {  
 **val** dep = dependencies.head.asInstanceOf[ShuffleDependency[K, V, C]]  
 SparkEnv.*get*.shuffleManager.getReader(dep.*shuffleHandle*, split.index, split.index + 1, context)  
 .read()  
 .asInstanceOf[Iterator[(K, C)]]  
}

1. SortShuffleManager中的源码如下：

**override def** getReader[K, C](  
 handle: ShuffleHandle,  
 startPartition: Int,  
 endPartition: Int,  
 context: TaskContext): ShuffleReader[K, C] = {  
 **new** BlockStoreShuffleReader(  
 handle.asInstanceOf[BaseShuffleHandle[K, \_, C]], startPartition, endPartition, context)  
}

1. BlockStoreShuffleReader：用来读取上游Task中计算的结果。具体是通过read方法实现，其源码如下：

其中ShuffleBlockFetcherIterator是用来从远程或者本地获取数据。

*/\*\* Read the combined key-values for this reduce task \*/***override def** read(): Iterator[Product2[K, C]] = {  
 **val** blockFetcherItr = **new** ShuffleBlockFetcherIterator(  
 context,  
 blockManager.*shuffleClient*,  
 blockManager,  
 mapOutputTracker.getMapSizesByExecutorId(handle.shuffleId, startPartition, endPartition),  
 // Note: we use getSizeAsMb when no suffix is provided for backwards compatibility  
 SparkEnv.*get*.conf.getSizeAsMb("spark.reducer.maxSizeInFlight", "48m") \* 1024 \* 1024)  
  
 // Wrap the streams for compression based on configuration  
 **val** wrappedStreams = blockFetcherItr.map { **case** (blockId, inputStream) =>  
 blockManager.wrapForCompression(blockId, inputStream)  
 }

1. 其中ShuffleBlockFetcherIterator在初始化的时候，会调用splitLocalRemoteBlocks()方法划分是从本地还是远程读取BlocksW

**private**[**this**] **def** initialize(): Unit = {  
 // Add a task completion callback (called in both success case and failure case) to cleanup.  
 context.addTaskCompletionListener(\_ => cleanup())  
  
 // Split local and remote blocks.  
 **val** remoteRequests = splitLocalRemoteBlocks()

// Add the remote requests into our queue in a random order  
 *fetchRequests* ++= Utils.*randomize*(remoteRequests)  
  
 // Send out initial requests for blocks, up to our maxBytesInFlight  
 fetchUpToMaxBytes()  
  
 **val** numFetches = remoteRequests.size - *fetchRequests*.size  
 logInfo("Started " + numFetches + " remote fetches in" + Utils.*getUsedTimeMs*(*startTime*))  
//调用fetchLocalBlocks从本地获取Blocks  
 // Get Local Blocks  
 fetchLocalBlocks()  
 logDebug("Got local blocks in " + Utils.*getUsedTimeMs*(*startTime*))  
}

例如：从本地获取

1. 是通过fetchLocalBlocks方法实现,其中是通过BlockManager中的getBlockData方法获取本地Block。

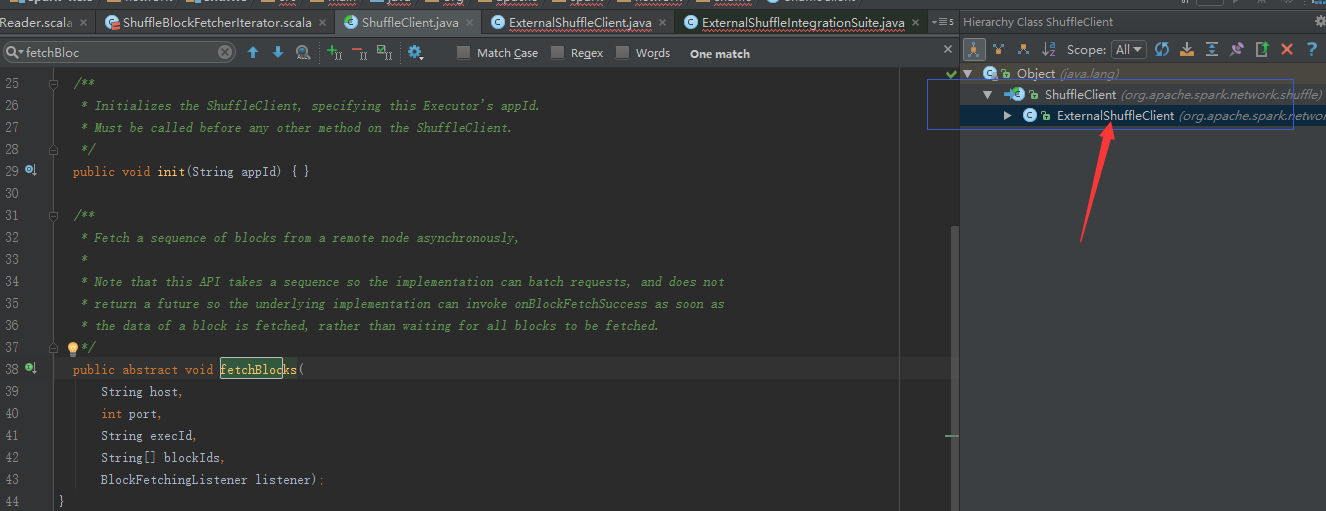
*/\*\*  
 \* Fetch the local blocks while we are fetching remote blocks. This is ok because  
 \** [[*ManagedBuffer*]]*'s memory is allocated lazily when we create the input stream, so all we  
 \* track in-memory are the ManagedBuffer references themselves.  
 \*/***private**[**this**] **def** fetchLocalBlocks() {  
 **val** iter = *localBlocks*.iterator  
 **while** (iter.hasNext) {  
 **val** blockId = iter.next()  
 **try** {  
 **val** buf = blockManager.getBlockData(blockId)  
 *shuffleMetrics*.incLocalBlocksFetched(1)  
 *shuffleMetrics*.incLocalBytesRead(buf.size)  
 buf.retain()  
 *results*.put(**new** SuccessFetchResult(blockId, blockManager.*blockManagerId*, 0, buf))  
 } **catch** {  
 **case** e: Exception =>  
 // If we see an exception, stop immediately.  
 logError(s"Error occurred while fetching local blocks", e)  
 *results*.put(**new** FailureFetchResult(blockId, blockManager.*blockManagerId*, e))  
 **return** }  
 }  
}

从远程获取Block

1. 通过sendRequest方法获取远程的Block。其内部是通过ShuffleClient的fetchBlocks方法来获取其他节点上的中间结果数据。

**private**[**this**] **def** sendRequest(req: FetchRequest) {  
 logDebug("Sending request for %d blocks (%s) from %s".format(  
 req.blocks.size, Utils.*bytesToString*(req.*size*), req.address.hostPort))  
 *bytesInFlight* += req.*size* // so we can look up the size of each blockID  
 **val** sizeMap = req.blocks.map { **case** (blockId, size) => (blockId.toString, size) }.toMap  
 **val** blockIds = req.blocks.map(\_.\_1.toString)  
  
 **val** address = req.address  
 shuffleClient.fetchBlocks(address.host, address.port, address.executorId, blockIds.toArray,  
 **new** BlockFetchingListener {  
 **override def** onBlockFetchSuccess(blockId: String, buf: ManagedBuffer): Unit = {  
 // Only add the buffer to results queue if the iterator is not zombie,  
 // i.e. cleanup() has not been called yet.  
 **if** (!*isZombie*) {  
 // Increment the ref count because we need to pass this to a different thread.  
 // This needs to be released after use.  
 buf.retain()  
 *results*.put(**new** SuccessFetchResult(*BlockId*(blockId), address, sizeMap(blockId), buf))  
 *shuffleMetrics*.incRemoteBytesRead(buf.size)  
 *shuffleMetrics*.incRemoteBlocksFetched(1)  
 }  
 logTrace("Got remote block " + blockId + " after " + Utils.*getUsedTimeMs*(*startTime*))  
 }  
  
 **override def** onBlockFetchFailure(blockId: String, e: Throwable): Unit = {  
 logError(s"Failed to get block(s) from **$**{req.address.host}:**$**{req.address.port}", e)  
 *results*.put(**new** FailureFetchResult(*BlockId*(blockId), address, e))  
 }  
 }  
 )  
}

1. 此fetchBlocks方法是ExternalShuffleClient中具体实现。



1. fetchBlocks方法的源码如下：

@Override  
public void fetchBlocks(  
 final String host,  
 final int port,  
 final String execId,  
 String[] blockIds,  
 BlockFetchingListener listener) {  
 checkInit();  
 logger.debug("External shuffle fetch from {}:{} (executor id {})", host, port, execId);  
 try {  
 RetryingBlockFetcher.BlockFetchStarter blockFetchStarter =  
 new RetryingBlockFetcher.BlockFetchStarter() {  
 @Override  
 public void createAndStart(String[] blockIds, BlockFetchingListener listener)  
 throws IOException {  
 TransportClient client = clientFactory.createClient(host, port);  
 new OneForOneBlockFetcher(client, appId, execId, blockIds, listener).start();  
 }  
 };  
  
 int maxRetries = conf.maxIORetries();  
 if (maxRetries > 0) {  
 // Note this Fetcher will correctly handle maxRetries == 0; we avoid it just in case there's  
 // a bug in this code. We should remove the if statement once we're sure of the stability.  
 new RetryingBlockFetcher(conf, blockFetchStarter, blockIds, listener).start();  
 } else {  
 blockFetchStarter.createAndStart(blockIds, listener);  
 }  
 } catch (Exception e) {  
 logger.error("Exception while beginning fetchBlocks", e);  
 for (String blockId : blockIds) {  
 listener.onBlockFetchFailure(blockId, e);  
 }  
 }  
}